hochschule mannheim

Subsumption-Architektur an einem Lego EV3-Roboter

Steffen Bartsch
Matr.Nr. 1515374

Robotik
Künstliche Intelligenz und autonome Systeme
Fakultät für Informatik
Hochschule Mannheim

06.07.2018

Dozenten
Prof. Dr. Thomas Ihme
Prof. Dr. Jörn Fischer

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ich bin damit einverstanden, dass meine Arbeit veröffentlicht wird, d. h. dass die Arbeit elektronisch gespeichert, in andere Formate konvertiert, auf den Servern der Hochschule Mannheim öffentlich zugänglich gemacht und über das Internet verbreitet werden darf.

Mannheim, 06.07.2018

Steffen Bartsch

Abstract

Subsumption-Architektur an einem Lego EV3-Roboter

Diese Arbeit zeigt die Umsetzung einer Subsumtion-Architektur an einem Lego EV3-Roboter. Das Projekt wurde in den Vorlesungen Robotik und autonome Systeme an der Hochschule Mannheim ausgearbeitet und umgesetzt. Das Konzept der, auf Schichten aufgebauten, Subsumtion-Architektur wird erklärt. Die verwendete Lego Hardware, wie EV3-Stein und Lego-Mindstorm-Sensoren, werden beschrieben. Weiterhin wird die Implementierung eines autonomen Roboters in Beispielen erklärt. Die Verhalten, die der Roboter unterscheidet, sind das Folgen einer Linie, Suchen einer Linie, Erkennen von Abbiegepunkten, Hindernisse umfahren, Beschleunigen an einer Rampe und sich selbst retten, wenn der Roboter die Tischkante erreicht. Die hieraus gewonnenen Kenntnisse werden beschrieben.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Zum Erlangen des akademischen Grades Bachelor of Science, habe ich die Fächer Robotik und Künstliche Intelligenz und autonome Systeme belegt. In Kombination der beiden Fächer möchte ich lernen wie man einem Roboter verschiedene Verhaltensweisen beibringen kann und wie ein Roboter sein Verhalten bestimmt.

1.2 Ziel der Arbeit

Diese Arbeit soll die Umsetzung einer Subsumption-Architektur an einem Lego EV3-Roboter für Studenten, welche die Vorlesung Robotik belegen, erklären. Der Lego EV3-Stein, im Zusammenhang mit LeJOS, wird beschrieben. Außerdem werden die allgemeinen Konzepte der Subsumption-Architektur erklärt und die Implementierung der Architektur im objektorientierten Umfeld aufgezeigt. Hierdurch soll ein Lego-Roboter weitestgehend in der Lage sein den Parcours der Robotik-Vorlesung autonom, durch Steuerung seiner Verhaltensweisen, zu bewältigen.

Kapitel 2

Architektur

2.1 Subsumption Architektur

Die Subsumption-Architektur ist eine reaktive Steuerungsarchitektur für autonome mobile Roboter, die erstmals 1985 von Rodney Brooks und seinen Kollegen am MIT vorgestellt wurde (Brooks 1985). Reaktiv bedeutet in diesem Zusammenhang, dass der Roboter keine interne Repräsentation seiner Umwelt zwischenspeichert, um daraus Handlungsanweisungen abzuleiten. Stattdessen werden alle nötigen Informationen zur Handlungssteuerung direkt aus den aktuellen Sensordaten gewonnen. [Schöbel, Leimbach und Jost 2014, S. 126]

Die Subsumption-Architektur bietet in der Robotik die Möglichkeit, einen echtzeitfähigen und reaktiven Roboter zu gestalten. Durch die Zentralisierung der Sensorauswertung im EV3-Brick ist es jedoch nicht möglich, mit leJOS ein echtzeitfähiges System aufzubauen. leJOS bietet dennoch die Möglichkeit, Roboter nach dem Vorbild der Subsumption-Architektur zu programmieren. Dazu wird im Package "lejos.robotics.subsumption" das Interface *Behavior* und die Klasse *Arbitrator* zur Verfügung gestellt. [Schöbel, Leimbach und Jost 2014, S. 128]

2.2 Schichtenmodell

Dieser Ansatz stand im Gegensatz zur traditionellen Robotikforschung, bei der stets auf eine symbolische Repräsentation der Umwelt zurückgegriffen wurde, und hatte großen Einfluss auf die Entwicklung autonomer Roboter. In der Subsumption Architektur wird das Gesamtverhalten eines Roboters durch verschiedene "Unter-

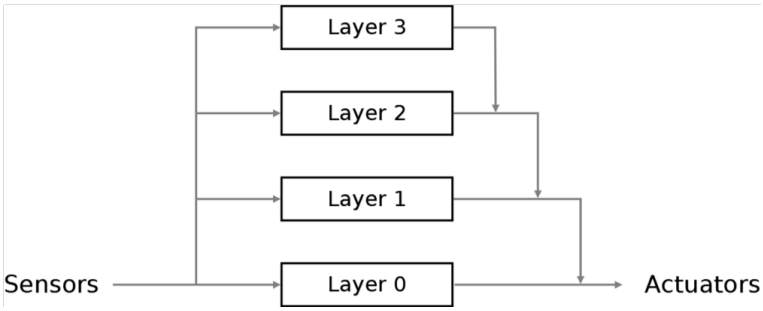

Abbildung 2.1: Schichtenmodell der Subsumption-Architektur

verhalten" gebildet. Diese "Unterverhalten" sind hierarchisch in Form von übereinander liegenden Schichten (engl.: Layer) organisiert, wobei jede Schicht einer bestimmten Aufgabe des Roboters entspricht. Dargestellt in Abbildung 2.1. Je höher die Schicht liegt, desto höher ist die Priorität. Das Layer0-Verhalten entspricht dem Standardverhalten und hat somit die geringste Priorität. [Schöbel, Leimbach und Jost 2014, S. 126-127]

Kapitel 3

Implementierung für Lego EV3

3.1 EV3 und Sensoren

EV3 bezeichnet die dritte Generation des LEGO MINDSTORMS-Systems und ist ein von LEGO eingetragenes Warenzeichen. Dabei handelt es sich um einen programmierbaren 32-bit ARM9-Mikrocontroller vom Hersteller Texas Instruments mit einem auf Linux basierenden Betriebssystem. Dieser wird auch EV3-Brick oder EV3-Stein genannt. Die Taktrate beträgt 300 MHz und die Größe des Hauptspeichers beträgt 64MB. Als Kommunikations- bzw. Programmierschnittstelle verfügt der EV3 über einen USB-2.0-Anschluss und ein Funkmodul das Bluetooth als Standard verwendet. Der Baustein verfügt über vier Motoranschlüsse (Port A bis D) und vier Sensoranschlüsse (Port 1 bis 4). [Schöbel, Leimbach und Jost 2014, S. 22]

Über die Ports können beispielsweise folgende Sensoren angeschlossen werden:

- **Farbsensor** - Der Farbsensor des EV3 ist ein digitaler Sensor, der vier verschiedene Modi unterstützt. Zum einen kann der Sensor im "Farb-ID" Modus Farben erkennen, zum anderen in den beiden Modi "Rotlicht" und "Umgebungslicht" die Lichtintensität messen. Darüberhinaus wird im "RGB"-Modus die Intensität der drei Grundfarben Rot, Grün und Blau gemessen.
- **Tastsensor** - Der Tastsensor ist ein sehr einfacher Sensor. Er kann lediglich erfassen, ob er gedrückt ist oder nicht.
- **Ultraschallsensor** - Der Ultraschallsensor kann die Distanz zu einem Objekt erkennen. Er misst die Distanz eines Objektes von 3cm bis 250cm mit einer Genauigkeit von +/- 1 cm.

- **Beschleunigungssensor** - Der Beschleunigungssensor misst die Beschleunigung auf drei Achsen in einem Bereich von -2 g / +2 g. Mit dem Sensor kann außerdem die Neigung gegenüber der Horizontalen ermittelt werden.
- **Gyrosensor** - Der Gyrosensor erkennt Drehbewegungen um seine eigene vertikale Achse mit einer Frequenz von 1 kHz.
- **Infrarotsensor** - Der digitale Infrarotsensor ist in der Lage, Infrarotlicht, das von Festkörperobjekten reflektiert wird, zu erkennen und so grob die Entfernung zu diesen abzuschätzen.

3.2 Lego Java-Operating-System

Bei Lego Java-Operating-System (leJOS) handelt es sich um eine schlanke Java Virtual Machine (JVM), die seit 2013 für den EV3-Stein verfügbar ist [lejos.org 2018]. leJOS begann ursprünglich als Hobby-Open-Source Projekt von José Solórzano im Jahr 1999 unter dem Namen TinyVM. Durch leJOS ist es möglich den EV3 mit Hilfe der Hochsprache Java zu programmieren. Ebenso trägt auch die Klassenbibliothek, mit der die Komponenten des EV3 (Motoren, Sensoren etc.) angesprochen werden, den Namen leJOS. Damit stellt die leJOS-Software ein Java-Betriebssystem als Alternative zur vorinstallierten LEGO-Firmware für den EV3 dar. leJOS wird von einer bootbaren microSD-Karte gestartet, ohne dabei die auf dem EV3 vorhandene LEGO-Software zu verändern bzw. zu löschen.

Durch die Erweiterung mit leJOS können die Vorteile der Objektorientierung für den EV3 genutzt werden. Es ist mit wenig Aufwand in professionelle Entwicklungsumgebungen integrierbar und arbeitet schneller und wesentlich ressourcenschonender als die grafische Lösung EV3-G. Zusätzlich bietet leJOS eine vollständige BluetoothUnterstützung und ist leicht erweiterbar, beispielsweise um Klassen für Sensoren von Drittherstellern zu nutzen. Des Weiteren unterstützt leJOS High-Level-Robotik-Tasks (Navigation, Localization etc.) und besitzt eine große Community (inklusive Wiki und dokumentierter API). leJOS stellt zudem ein vollständiges Java-Runtime-System zur Verfügung. In in der Entwicklungsumgebung von Eclipse kann leJOS sehr einfach integriert werden. Es muss lediglich die Bibliothek dem Java-Build-Path zugefügt werden. [Schöbel, Leimbach und Jost 2014, S. 23]

3.3 Behavior

Behavior [lejos.org 2018] wird benutzt, um verschiedene Verhaltensweisen des Roboters zu beschreiben. Mit Hilfe des *Behavior*-Interfaces können eigene Klassen definiert werden, bei der jede Klasse einem bestimmten Verhalten (engl.: behavior) entspricht. Das Interface *Behavior* gibt dabei nur den Namen und die Art der Methoden vor. Eine abgeleitete Klasse muss dann die folgenden Methoden implementieren [Schöbel, Leimbach und Jost 2014, S. 128 - 129]:

- **action()** - beschreibt die Aktionen, die der Roboter ausführt, wenn dieses Verhalten aktiv wird. Dies kann beispielsweise nur das Abspielen eines Tons sein, oder etwas Komplexeres, wie die Navigation in einem Raum. Es muss berücksichtigt werden, dass die Methode beendet werden muss, sobald die im Folgenden beschriebene Methode suppress() aufgerufen wird. Die Ausführung oder Unterdrückung der Methode kann einfach über eine Boolean-Variable kontrolliert werden. Diese wird gesetzt, sobald suppress() aufgerufen wird.
- **suppress()** - Der Aufruf dieser Methode muss das aktuelle Verhalten schnellstmöglich beenden. Eventuell gestartete Threads müssen beendet und die Methode action() zum Auslaufen gebracht werden.
- **takeControl()** - prüft, ob das beschriebene Verhalten die Kontrolle über den Roboter übernehmen und ein entsprechender boolean-Wert zurückgegeben werden sollte. Eine einfache Implementierung könnte beispielsweise die Abfrage eines Touch-Sensors sein, der prüft ob ein Objekt berührt wurde und ein entsprechendes Ausweich-Verhalten aktiviert werden muss.

3.4 Arbitrator

Der *Arbitrator* [lejos.org 2018] ist dafür zuständig die Hierarchie zu realisieren, und die Verhalten zu aktivieren sowie zu deaktivieren. Instanziiert wird ein *Arbitrator*-Objekt mit einer Liste von *Behavior*-Objekten, die die verschiedenen Verhalten des Roboters darstellen. Der Index des Behavior-Objekts in der Liste entspricht der Priorität des Verhaltens. Das Verhalten mit dem größten Index hat also auch die höchste Priorität. Ist der *Arbitrator* einmal instanziiert, können die ihm übergebenenen Verhalten nicht mehr verändert werden. Nach der Instanziierung muss die

Abfrage und Steuerung der Verhalten durch den *Arbitrator* noch mit der Methode start() gestartet werden.

Die Instanziierung des *Arbitrators* kann mit folgender Parameterliste (Behavior[] behaviorList, boolean returnWhenInactive) durchgeführt werden. Das Array behaviorList enthält die Verhalten, welche der *Arbitrator* überprüft und steuert. Der Parameter returnWhenInactive ist optional und ist standardmäßig auf *false*, was bezweckt, dass die start()-Methode in einer Endlosschleife läuft und immer wieder das Layer0-Verhalten angenommen wird. Diesen Parameter auf *true* zu setzen bietet sich an, um die Verhalten zu testen. Hierdurch wird jedes Verhalten nur einmal vom *Arbitrator* ausgeführt und die start()-Methode terminiert danach. [Schöbel, Leimbach und Jost 2014, S. 129]

3.5 Objektorientiertes Modell

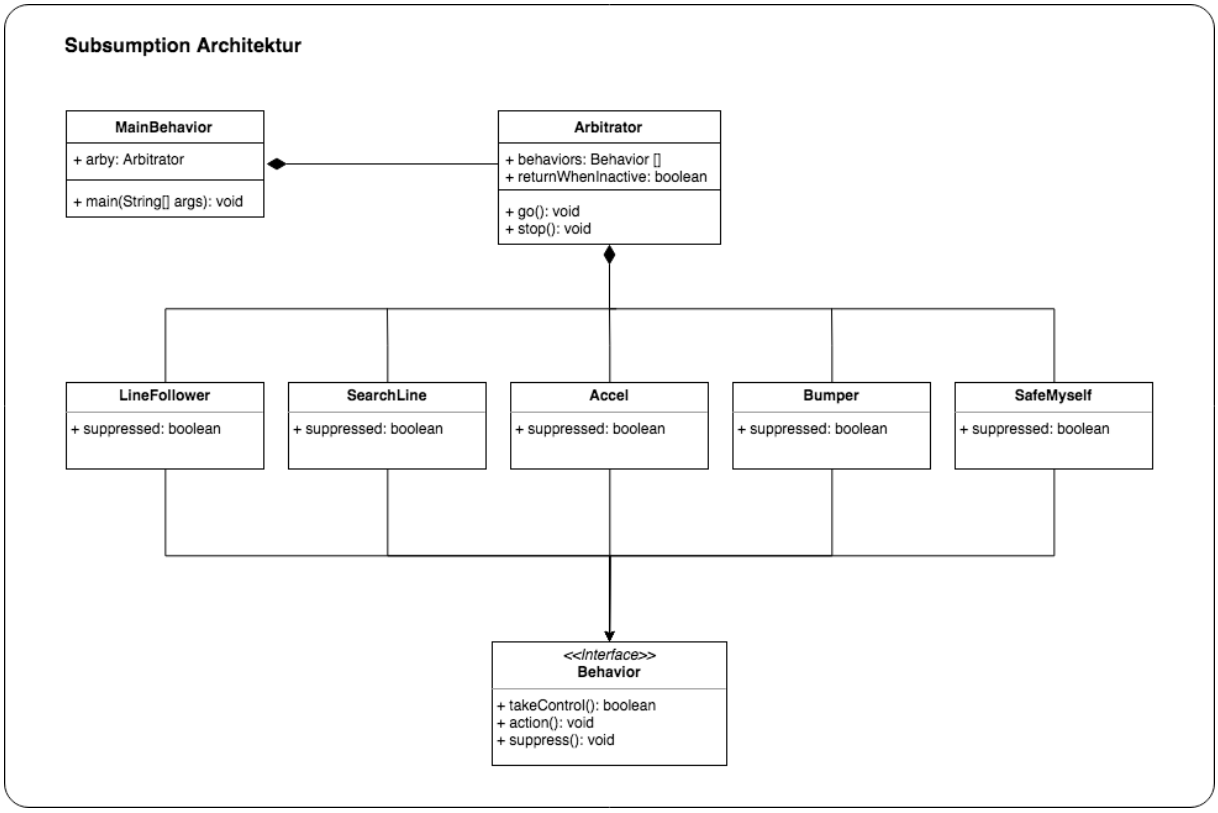

Abbildung 3.1: Klassendiagramm der Subsumption-Architektur

Das in Abbildung 3.1 gezeigte Diagramm beschreibt die Klassen für die objektorientierte Implementierung der Subsumption-Architektur. Die Klasse MainBehavior

startet das Programm und enthält den den *Arbitrator*, der die verschiedenen Verhalten prüft und kontrolliert steuert. Die Klassen LineFollower, SearchLine, Accel, Bumper und SafemMySelf implementieren das Interface Behavior.

In Abbildung 3.2 wird die Anordnung der Behaviorklassen nach der Priorität verdeutlicht, wobei LineFollower hier die geringste Prorität hat und somit das Standardverhalten implementiert.

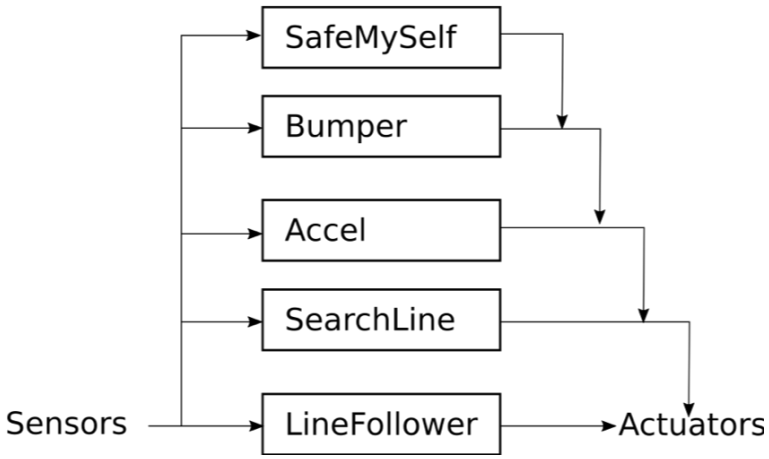

Abbildung 3.2: Schichtenmodell anhand der implementierten Verhalten

3.5.1 Linienfolger

Das Verhalten Linienfolger (engl.: Line Follower), bildet das Standardverhalten und prüft über die beiden Farbsensoren welche Farben sich vor dem Roboter befinden. Für den Parcours müssen die Farben Weiß, Schwarz oder Grün unterschieden werden. Die Farben werden als Integer im Quellcode, in einer Skala von -1 bis 9, dargestellt. Der Integer -1 symbolisiert, dass der Farbsensor keinen definierten Farbwert erkennt. Wird auf beiden Sensoren die Farbe weiß erkannt, fährt der Roboter gerade aus. Wenn der Roboter auf einer Seite Weiß und auf der anderen Seite Schwarz erkennt. Dreht der Roboter bis wieder auf beiden Sensoren Weiß erkannt wird. Wenn der Roboter auf einem der beiden Sensoren Grün erkennt, biegt er in Richtung des grünen Puktes ab und orientiert sich dann wieder an der schwarzen Linie. Dieses Verhalten kann durch Drücken des Knopfes mit der *ID UP* manuell unterdrückt werden.

3.5.2 Linie suchen

Dieses Verhalten prüft, über die Radumdrehung, wie weit der Roboter gefahren ist. Solange die Farbsensoren nur Weiß erkennen, wird der Winkel für die Radumdrehung aufaddiert. Sobald einer der beiden Farbsensoren Schwarz erkennt, wird der Zähler für die Radumdrehung zurückgesetzt. Wird die festgelegte Distanz erreicht, nimmt der Roboter das *SearchLine*-Verhalten an, z. B. wenn er die Linie an einer Unterbrechung verloren hat. Hierbei fährt er abwechselnd, ebenfalls für eine festgelegte Distanz, nach links und rechts und führt dabei die Farberkennung und die Messung der Radumdrehung aus. Sobald er wieder eine schwarze Linie gefunden hat oder die Tischkante erreicht, wird das Suchverhalten sofort unterdrückt.

3.5.3 Hindernis Ausweichen

Das Ausweichen eines Hindernisses erfolgt durch die Betätigung des Tastsensors. Der Tastsensor meldet die Integerwerte 0 und 1. Meldet der Sensor eine 1, wird das Ausweichverhalten aktiviert und der Wert wieder auf 0 gesetzt. Der Roboter fährt dann ein kleines Stück zurück und führt einen einfachen sequenziellen Algorithmus aus, der ihn um das Hindernis bewegt. Nach diesem Ablauf wird das Verhalten unterdrückt.

3.5.4 Beschleunigen an der Rampe

Der Roboter verwendet einen Beschleunigungssensor um festzustellen, ob der Roboter eine Neigung erreicht. Wird eine festgelegte Neigung erreicht, beschleunigt der Roboter die beiden Antriebsmotoren um die Rampe hochzufahren. Erkennt der Beschleunigungssensor auf der Rampe wieder die gerade Ebene, werden die Antriebsmotoren wieder gebremst.

3.5.5 Retten an der Tischkante

Das Ende der Tischkante wird mit den Farbsensoren festgestellt. Die Farbsensoren melden den Wert -1, wenn der Roboter die Tischkante erreicht. Hierdurch wird das *SafeMyself*-Verhalten ausgelöst. Der Roboter fährt eine festgelegte Distanz zurück und beginnt sich auf der Stelle zu drehen und dabei die Farbsensoren abzutasten.

Erkennt er eine schwarze Linie, wird das verhalten unterdrückt und der Roboter folgt wieder der Linie.

3.6 MainBehavior

Die *MainBehavior*-Klasse ist übergeordnet für die *Behavior*-Klassen. Hier werden die verschiedenen Verhalten und der *Arbitrator* erzeugt. Da die Farbsensoren in verschiedenen Verhalten parallel verwendet werden, werden sie ebenfalls in der *MainBehavior*-Klasse initialisiert und beim erzeugen der Verhaltensobjekte als Parameter an den jeweiligen Konstruktor übergeben. Die Standardgeschwindigkeit wird hier gesetzt und dann der Arbitrator gestartet. Der Arbitrator regelt die Steuerung der Verhalten und läuft dann, solange *returnWhenInactive* nicht gesetzt ist, in einer Endlosschleife.

Kapitel 4

Erkenntnis

Während der Programmierung der Subsumption Architektur, an dem EV3-Roboter, konnte ich die im Nachfolgenden erklärten Erfahrungen sammeln. Die Umsetzung der Subsumption Architektur an einem Lego EV3-Roboter ist recht einfach. Von leJOS werden Bereits die nötigen Klassen und Schnittstellen geliefert, so dass der Roboter mit einem Arbitrator-Objekt die Sensoren parallel abtastet und die Aktivierung der Verhalten steuert.

Da die Verhalten nach aufsteigender Priorität gesetzt werden, muss berücksichtigt werden welches Verhalten weiterhin geprüft werden soll, sobald sich der Roboter in einem höherwertigem Verhalten befindet, z. B. soll der Roboter weiterhin erkennen ob er eine Absturzkante erreicht, wenn er an der Rampe beschleunigt.

Ich bin auf folgende technische Probleme gestoßen. Ein initiales Problem war die Verwendung von den Farbsensoren in mehreren parallelen Threads. Da die Ports des EV3 nur einmal angesprochen werden dürfen, muss dies in der *MainBehavior*-Klasse geschehen. Der *SensorMode* kann dann an die jeweiligen Verhalten im Konstruktor übergeben werden. Außerdem ist die Abtastrate oft zu langsam. Gerade an der Tischkante kann die lange Dauer für die Abtastung des Farbsensors gefährlich werden. Ein weiteres Problem ist eine willkürliche Beschleunigung der Antriebsmotoren. Es kommt in unterschiedlichen Situationen vor, dass der Roboter kurz einen Antriebsmotor beschleunigt, was gerade an der Tischkante sehr kritisch werden kann. Dieses Problem konnte im Laufe der Vorlesung nicht gelöst werden. Während der Ausweichmethode konnte die Breite des Hindernisses nicht abgefragt werden. Hier wäre ein weiterer Port für einen Infrarot- oder Ultraschallsensor sehr praktisch gewesen. Auch wenn der Winkel, in dem der Roboter an das Hindernis fährt, zu schräg ist, wird der Tastsensor nicht betätigt und der Roboter schiebt das

Hindernis vor sich her. Das Hoch- und Runterfahren an der Rampe hat sich auch als Herausforderung herausgestellt. Nur mit Raupenantrieb und Gewichtsverteilung war es möglich die Rampe hochzufahren und beim Runterfahren nicht nach vorne umzukippen. Hierzu war es auch nötig, eine bewegliche Aufhängung für die Farbsensoren zu montieren, um eine Blockade durch starr montierte Sensoren zu vermeiden. Da die Motoren nicht ganz gleich Beschleunigen, mussten die Motoren mit verschiedenen Beschleunigungswerten initialisiert werden um an der Rampe nicht seilich abzudriften.

Schwierigkeiten machte aber auch die Programmierung der Verhaltensalgorithmen. Beim Suchen der Linie braucht man einen Algorithmus der den Roboter bei erfolglosem Suchen, möglichst wieder an seinen Ausgangspunkt zurückführt. Ein weiteres Problem ist aufgetreten, wenn der Roboter dann in einem ca. 90 Grad Winkel auf eine gefundene Linie trifft. Wenn auf beiden Farbsensoren Schwarz erkannt wird, verhält sich der Roboter wie an einer Kreuzung und fährt geradeaus weiter. Auch wenn wieder eine Linie gefunden wird und der Roboter dieser wieder folgt, weiß er nicht welche Richtung die richtige ist.

Trotz der Probleme konnte der Roboter recht sicher der Linie folgen, er konnte sicher Abbiegen wenn grüne Punkte erkannt wurden, wenn er die Linie verloren hatte konnte er sich selbst wieder einordnen, er konnte das Hindernis umfahren, hat die Rampe überwunden ohne umzufallen und hat die Tischkante erkannt und sich dann selbst gerettet. Mir wurde bewusst wie wichtig das Testen bei der Roboterprogrammierung ist. Dies braucht viel Zeit, da das Verhalten des Roboters nach jeder Änderung im Quellcode getestet werden muss.

Abkürzungsverzeichnis

leJOS Lego Java-Operating-System

JVM Java Virtual Machine

Abbildungsverzeichnis

Literatur

lejos.org (2018). *leJOS API Dokumentation*. URL: http://www.lejos.org/ev3.php (besucht am 30. 06. 2018).

Schöbel, Maximilian, Torsten Leimbach und Beate Jost (2014). *Roberta EV3-Programmieren mit Java*. Februar 2015. ISBN (Print) 978-3-8396-0840-1. Frauenhofer Verlag. URL: https://www.roberta-home.de/fileadmin/user_upload/Roberta-EV3 programmierenJava_small.pdf.